Script generated by TTT

Title: Petter: Programmiersprachen (28.10.2015)

Date: Wed Oct 28 14:18:56 CET 2015

Duration: 96:17 min

Pages: 47

Example: The Dekker Algorithm on SMP Systems

Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of two processes with busy waiting.

```
//flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn    = 0   // or 1
```

```
P0:
flag[0] = true;
while (flag[1] == true)
  if (turn != 0) {
     flag[0] = false;
     while (turn != 0) {
       // busy wait
     }
     flag[0] = true;
  }
// critical section
turn    = 1;
flag[0] = false;
```


Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of two processes with busy waiting.

```
//flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn    = 0   // or 1
```

```
P0:
flag[0] = true;
while (flag[1] == true)
  if (turn != 0) {
     flag[0] = false;
     while (turn != 0) {
       // busy wait
     }
     flag[0] = true;
  }
// critical section
turn    = 1;
flag[0] = false;
```


```
P1:
flag[1] = true;
while (flag[0] == true)
  if (turn != 1) {
     flag[1] = false;
     while (turn != 1) {
       // busy wait
     }
     flag[1] = true;
  }
// critical section
turn    = 0;
flag[1] = false;
```

The Idea Behind Dekker

Communication via three variables:

- `flag[i]=true` process P_i wants to enter its critical section
- `turn=i` process P_i has priority when both want to enter

```
P0:
flag[0] = true;
while (flag[1] == true)
  if (turn != 0) {
      flag[0] = false;
      while (turn != 0) {
        // busy wait
      }
      flag[0] = true;
  }
// critical section
turn     = 1;
flag[0] = false;
```

In process P_i:

- if P_{1-i} does not want to enter, proceed immediately to the critical section

The Idea Behind Dekker

Communication via three variables:

- `flag[i]=true` process P_i wants to enter its critical section
- `turn=i` process P_i has priority when both want to enter

```
P0:
flag[0] = true;
while (flag[1] == true)
  if (turn != 0) {
      flag[0] = false;
      while (turn != 0) {
        // busy wait
      }
      flag[0] = true;
  }
// critical section
turn     = 1;
flag[0] = false;
```

In process P_i:

- if P_{1-i} does not want to enter, proceed immediately to the critical section
- ⇝ `flag[i]` is a *lock* and may be implemented as such
- if P_{1-i} also wants to enter, wait for `turn` to be set to `i`

The Idea Behind Dekker

Communication via three variables:

- `flag[i]=true` process P_i wants to enter its critical section
- `turn=i` process P_i has priority when both want to enter

```
P0:
flag[0] = true;
while (flag[1] == true)
  if (turn != 0) {
      flag[0] = false;
      while (turn != 0) {
        // busy wait
      }
      flag[0] = true;
  }
// critical section
turn     = 1;
flag[0] = false;
```

In process P_i:

- if P_{1-i} does not want to enter, proceed immediately to the critical section
- ⇝ `flag[i]` is a *lock* and may be implemented as such
- if P_{1-i} also wants to enter, wait for `turn` to be set to `i`
- while waiting for `turn`, reset `flag[i]` to enable P_{1-i} to progress
- algorithm only works for two processes

Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of two processes with busy waiting.

```
//flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn    = 0   // or 1
```

```
P0:
flag[0] = true;
while (flag[1] == true)
  if (turn != 0) {
      flag[0] = false;
      while (turn != 0) {
        // busy wait
      }
      flag[0] = true;
  }
// critical section
turn     = 1;
flag[0] = false;
```

```
P1:
flag[1] = true;
while (flag[0] == true)
  if (turn != 1) {
      flag[1] = false;
      while (turn != 1) {
        // busy wait
      }
      flag[1] = true;
  }
// critical section
turn     = 0;
flag[1] = false;
```

A Note on Dekker's Algorithm

Dekker's algorithm has the three desirable properties:

- *ensure mutual exclusion*: at most one process executes the critical section
- *deadlock free*: the process will never wait for each other
- *free of starvation*: if a process wants to enter, it eventually will

applications for Dekker: implement a (*map* ∘ *reduce+map*) operation concurrently

```
T acc = init();
for (int i = 0; i<c; i++) {
  <T,U> (acc,tmp) = f(acc,i); // read from inp[i]
  g(tmp, i);                  // write to out[i]
}
```

- accumulating a value by performing two operations `f` and `g` in sequence
- the calculation in `f` of the ith iteration depends on iteration $i-1$
- non-trivial program to parallelize
- idea: use two threads, one for `f` and one for `g`

Concurrent Reduce+Map

Create an n-place buffer for communication between processes P_f and P_g.

```
T acc = init();
Buffer<U> buf = buffer<T>(n); // some locked buffer
```

```
Pf:
for (int i = 0; i<c; i++) {
  <T,U> (acc,tmp) = f(acc,i);
  buf.put(tmp);
}
```

```
Pg:
for (int i = 0; i<c; i++) {
  T tmp = buf.get();
  g(tmp, i);
}
```


Dekker's Algorithm and Weakly-Ordered

Problem: Dekker's algorithm requires sequentially consistency.
Idea: insert memory barriers between all variables common to both threads.

Dekker's Algorithm and Weakly-Ordered

Problem: Dekker's algorithm requires sequentially consistency.
Idea: insert memory barriers between all variables common to both threads.

```
P0:
flag[0] = true;
sfence();
while (lfence(), flag[1] == true)
  if (lfence(), turn != 0) {
    flag[0] = false;
    sfence();
    while (lfence(), turn != 0) {
      // busy wait
    }
    flag[0] = true;
    sfence();
  }
// critical section
turn    = 1;
sfence();
flag[0] = false; sfence();
```

- insert a load memory barrier `lfence()` in front of every read from common variables

Dekker's Algorithm and Weakly-Ordered

Problem: Dekker's algorithm requires sequentially consistency.
Idea: insert memory barriers between all variables common to both threads.

```
P0:
flag[0] = true;
sfence();
while (lfence(), flag[1] == true)
  if (lfence(), turn != 0) {
      flag[0] = false;
      sfence();
      while (lfence(), turn != 0) {
        // busy wait
      }
      flag[0] = true;
      sfence();
  }
// critical section
turn     = 1;
sfence();
flag[0] = false; sfence();
```

- insert a load memory barrier `lfence()` in front of every read from common variables
- insert a write memory barrier `sfence()` after writing a variable that is read in the other thread
- the `lfence()` of the first iteration of each loop may be combined with the preceding `sfence()` to an `mfence()`

Discussion

Memory barriers reside at the lowest level of synchronization primitives.

Discussion

Memory barriers reside at the lowest level of synchronization primitives.
Where are they useful?

- when several processes implement an automaton and ...
- synchronization means coordinating transitions of these automata
- when blocking should not de-schedule threads
- often used in operating systems

Why might they not be appropriate?

- difficult to get right, possibly inappropriate except for specific, proven algorithms
- often synchronization with locks is as fast and easier
- too many fences are costly if store/invalidate buffers are bottleneck

Discussion

Memory barriers reside at the lowest level of synchronization primitives.
Where are they useful?

- when several processes implement an automaton and ...
- synchronization means coordinating transitions of these automata
- when blocking should not de-schedule threads
- often used in operating systems

Why might they not be appropriate?

- difficult to get right, possibly inappropriate except for specific, proven algorithms
- often synchronization with locks is as fast and easier
- too many fences are costly if store/invalidate buffers are bottleneck

What do compilers do about barriers?

- C/C++: it's up to the programmer, use `volatile` for all thread-common variables to avoid optimizations which are only correct for sequential programs
- C++11: use of *atomic* variables will insert memory barriers
- Java,Go,...: the runtime system must guarantee this

Summary

Memory consistency models:

- strict consistency
- sequential consistency
- weak consistency

Illustrating consistency:

- happened-before relation
- happened-before process diagrams

Intricacy of cache coherence protocols:

- the effect of store buffers
- the effect of invalidate buffers
- the use of memory barriers

Use of barriers in synchronization algorithms:

- Dekker's algorithm
- stream processing, avoidance of busy waiting
- inserting fences

Future Many-Core Systems: NUMA

Symmetric multi-processing (SMP) has its limits:

- a memory-intensive computation may cause contention on the bus
- the speed of the bus is limited since the electrical signal has to travel to all participants
- point-to-point connections are faster than a bus, but do not provide possibility of forming consensus

Future Many-Core Systems: NUMA

Symmetric multi-processing (SMP) has its limits:

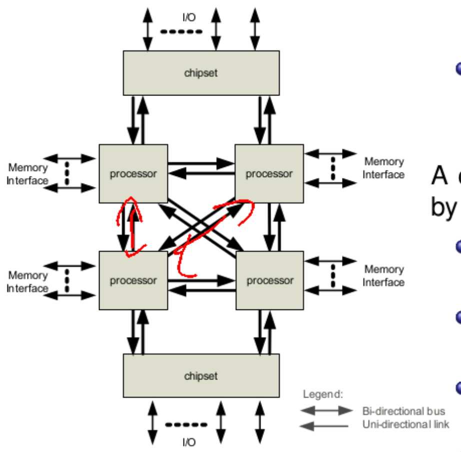
- a memory-intensive computation may cause contention on the bus
- the speed of the bus is limited since the electrical signal has to travel to all participants
- point-to-point connections are faster than a bus, but do not provide possibility of forming consensus

⇝ use a bus locally, use point-to-point links globally: *NUMA*

- *non-uniform memory access* partitions the memory amongst CPUs
- a directory states which CPU holds a memory region
- Intel's *MESIF* to reduce communication overhead Forward
- Interprocess communication between Cache-Controllers (*ccNUMA*): onchip on Opteron or in chipset on Itanium

Overhead of NUMA Systems

Communication overhead in a NUMA system.



source: [Int09]

- Processors in a NUMA system may be fully or partially connected.
- The directory of who stores an address is partitioned amongst processors.

A cache miss that cannot be satisfied by the local memory at A:

- A sends a retrieve request to processor B owning the directory
- B tells the processor C who holds the content
- C sends data (or status) to A and sends acknowledge to B
- B completes transmission by an acknowledge to A

References

Intel.
An introduction to the intel quickpath interconnect.
Technical Report 320412, 2009.

Leslie Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.
Commun. ACM, 21(7):558–565, July 1978.

Paul E. McKenny.
Memory Barriers: a Hardware View for Software Hackers.
Technical report, Linux Technology Center, IBM Beaverton, June 2010.

Scott Owens, Susmit Sarkar, and Peter Sewell.
A better x86 memory model: x86-TSO.
Technical Report UCAM-CL-TR-745, University of Cambridge, Computer Laboratory, March 2009.

Future Many-Core Systems: NUMA

Symmetric multi-processing (SMP) has its limits:

- a memory-intensive computation may cause contention on the bus
- the speed of the bus is limited since the electrical signal has to travel to all participants
- point-to-point connections are faster than a bus, but do not provide possibility of forming consensus

⇝ use a bus locally, point-to-point links globally: *NUMA*

- *non-uniform memory access* partitions the memory amongst CPUs
- a directory states which CPU holds a memory region
- Intel's *MESIF* to reduce communication overhead
- Interprocess communication between Cache-Controllers (*ccNUMA*): onchip on Opteron or in chipset on Itanium

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Concurrency: Atomic Executions, Locks and Monitors

Mi 02.07.014

Dr. Michael Petter
Winter term 2015

Why Memory Barriers are not Enough

Communication via memory barriers has only specific applications:

- coordinating state transitions between threads
- for systems that require minimal overhead (and no de-scheduling)

Often certain pieces of memory may only be modified by one thread at once.

- can use barriers to implement automata that ensure *mutual exclusion*
- ⇝ generalize the re-occurring concept of enforcing mutual exclusion

Why Memory Barriers are not Enough

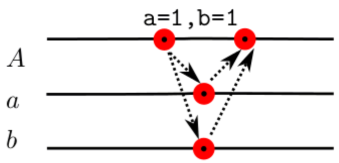
Communication via memory barriers has only specific applications:

- coordinating state transitions between threads
- for systems that require minimal overhead (and no de-scheduling)

Often certain pieces of memory may only be modified by one thread at once.

- can use barriers to implement automata that ensure *mutual exclusion*
- ⇝ generalize the re-occurring concept of enforcing mutual exclusion

Need a mechanism to update these pieces of memory as a single *atomic execution*:



- several values of the objects are used to compute new value
- certain information from the thread flows into this computation
- certain information flows from the computation to the thread

Atomic Executions

A concurrent program consists of several threads that share common resources:

- resources are often pieces of memory, but may be an I/O entity
  - a file can be modified through a shared handle
- for each resource an *invariant* must be retained
  - a head and tail pointer must define a linked list
- an invariant may span *several* resources
- during an update, an invariant may be *broken*

⇝ several resources must be updated together to ensure the invariant

- which particular resources need to be updated may depend on the current program state

Atomic Executions

A concurrent program consists of several threads that share common resources:

- resources are often pieces of memory, but may be an I/O entity
  - a file can be modified through a shared handle
- for each resource an *invariant* must be retained
  - a head and tail pointer must define a linked list
- an invariant may span *several* resources
- during an update, an invariant may be *broken*

⇝ several resources must be updated together to ensure the invariant

- which particular resources need to be updated may depend on the current program state

Ideally, we would want to mark a sequence of operations that update shared resources for *atomic execution* [Harris et al.(2010)Harris, Larus, and Rajwar]. This would ensure that the invariant never seems to be broken.

Overview

We will address the *established* ways of managing synchronization.

- present techniques are available on most platforms
- likely to be found in most existing (concurrent) software
- techniques provide solutions to solve common concurrency tasks
- techniques are the source of common concurrency problems

Presented techniques applicable to C, C++ (pthread), Java, C# and other imperative languages.

Overview

We will address the *established* ways of managing synchronization.

- present techniques are available on most platforms
- likely to be found in most existing (concurrent) software
- techniques provide solutions to solve common concurrency tasks
- techniques are the source of common concurrency problems

Presented techniques applicable to C, C++ (pthread), Java, C# and other imperative languages.

Learning Outcomes

1. Principle of Atomic Executions
2. Wait-Free Algorithms based on Atomic Operations
3. Locks: Mutex, Semaphore, and Monitor
4. Deadlocks: Concept and Prevention

Atomic Execution: Varieties

Definition (Atomic Execution)

A computation forms an *atomic execution* if its effect can only be observed as a single transformation on the memory.

Atomic Execution: Varieties

Definition (Atomic Execution)

A computation forms an *atomic execution* if its effect can only be observed as a single transformation on the memory.

Several classes of atomic executions exist:

Wait-Free : an atomic execution always succeeds and never blocks
Lock-Free : an atomic execution may fail but never blocks
Locked : an atomic execution always succeeds but may block the thread
Transaction : an atomic execution may fail (and may implement recovery)

Atomic Execution: Varieties

Definition (Atomic Execution)

A computation forms an *atomic execution* if its effect can only be observed as a single transformation on the memory.

Several classes of atomic executions exist:

Wait-Free : an atomic execution always succeeds and never blocks
Lock-Free : an atomic execution may fail but never blocks
Locked : an atomic execution always succeeds but may block the thread
Transaction : an atomic execution may fail (and may implement recovery)

These classes differ in

amount of data they can access during an atomic execution
expressivity of operations they allow
granularity of objects in memory they require

Wait-Free Atomic Executions

Wait-Free Updates

Which operations on a CPU are atomic executions? (`j` and `tmp` are registers)

Program 1

```
i++;
```

Program 2

```
j = i;
i = i+k;
```

Program 3

```
int tmp = i;
i = j;
j = tmp;
```


Wait-Free Updates

Which operations on a CPU are atomic executions? (`j` and `tmp` are registers)

Program 1

```
i++;
```

Program 2

```
j = i;
i = i+k;
```

Program 3

```
int tmp = i;
i = j;
j = tmp;
```

Answer:

- none by default (even without store and invalidate buffers, *why?*)
- but all of them *can* be atomic executions

Wait-Free Updates

Which operations on a CPU are atomic executions? (`j` and `tmp` are registers)

Program 1

```
i++;
```

Program 2

```
j = i;
i = i+k;
```

Program 3

```
int tmp = i;
i = j;
j = tmp;
```

Answer:

- none by default (even without store and invalidate buffers, *why?*)
- but all of them *can* be atomic executions

The programs can be atomic executions:

- `i` must be in memory (e.g. declared as volatile)
- Idea: *lock* the cache/bus for an adress for the duration of an instruction; on x86:
  - Program 1 can be implemented using a `lock inc [addr_i]` instruction
  - Program 2 can be implemented using `mov eax,k; lock xadd [addr_i],eax; mov reg_j,eax`
  - Program 3 can be implemented using `lock xchg [addr_i],reg_j`

Wait-Free Updates

Which operations on a CPU are atomic executions? (`j` and `tmp` are registers)

Program 1

```
i++;
```

Program 2

```
j = i;
i = i+k;
```

Program 3

```
int tmp = i;
i = j;
j = tmp;
```

Answer:

- none by default (even without store and invalidate buffers, *why?*)
- but all of them *can* be atomic executions

The programs can be atomic executions:

- `i` must be in memory (e.g. declared as volatile)
- Idea: *lock* the cache/bus for an adress for the duration of an instruction; on x86:
  - Program 1 can be implemented using a `lock inc [addr_i]` instruction
  - Program 2 can be implemented using `mov eax,k; lock xadd [addr_i],eax; mov reg_j,eax`
  - Program 3 can be implemented using `lock xchg [addr_i],reg_j`

⚠ Without `lock`, the load and store generated by `i++` may be interleaved with a store from another processor.

Wait-Free Bumper-Pointer Allocation

Garbage collectors often use a *bumper pointer* to allocated memory:

Bumper Pointer Allocation

```
char heap[2^20];
char* firstFree = &heap[0];

char* alloc(int size) {
  char* start = firstFree;
  firstFree = firstFree + size;
  if (start+size>sizeof(heap)) garbage_collect();
  return start;
}
```

- `firstFree` points to the first unused byte
- each allocation reserves the next `size` bytes in `heap`

Wait-Free Bumper-Pointer Allocation

Garbage collectors often use a *bumper pointer* to allocated memory:

Bumper Pointer Allocation

```
char heap[2^20];
char* firstFree = &heap[0];

char* alloc(int size) {
  char* start = firstFree;
  firstFree = firstFree + size;
  if (start+size>sizeof(heap)) garbage_collect();
  return start;
}
```

- `firstFree` points to the first unused byte
- each allocation reserves the next `size` bytes in `heap`

Thread-safe implementation:

- the `alloc` function can be used from multiple threads when implemented using a `lock xadd [_firstFree],eax` instruction
- ⇝ requires inline assembler

Marking Statements as Atomic

Rather than writing assembler: use made-up keyword `atomic`:

Program 1

```
atomic {
  i++;
}
```

Program 2

```
atomic {
  j = i;
  i = i+k;
}
```

Program 3

```
atomic {
  int tmp = i;
  i = j;
  j = tmp;
}
```

Marking Statements as Atomic

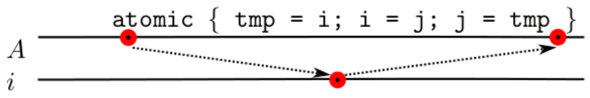
Rather than writing assembler: use made-up keyword `atomic`:

Program 1
```
atomic {
  i++;
}
```

Program 2
```
atomic {
  j = i;
  i = i+k;
}
```

Program 3
```
atomic {
  int tmp = i;
  i = j;
  j = tmp;
}
```

The statements in an `atomic` block execute as *atomic execution*:



- `atomic` only translatable when a corresponding atomic CPU instruction exist
- the notion of requesting *atomic execution* is a general concept

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:

- no control flow possible, no behavioral change depending on data
- often, there are instructions that execute an operation conditionally

Program 4
```
atomic {
  r = b;
  b = 0;
}
```

Program 5
```
atomic {
  r = b;
  b = 1;
}
```

Program 6
```
atomic {
  r = (k==i);
  if (r) i = j;
}
```

Operations *update* a memory cell and *return* the previous value.

- the first two operations can be seen as setting a flag `b` to $v \in \{0, 1\}$ if `b` not already contains v
  - this operation is called *modify-and-test*
- the third case generalizes this to arbitrary values
  - this operation is called *compare-and-swap*

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:

- no control flow possible, no behavioral change depending on data
- often, there are instructions that execute an operation conditionally

Program 4
```
atomic {
  r = b;
  b = 0;
}
```

Program 5
```
atomic {
  r = b;
  b = 1;
}
```

Program 6
```
atomic {
  r = (k==i);
  if (r) i = j;
}
```

Operations *update* a memory cell and *return* the previous value.

- the first two operations can be seen as setting a flag `b` to $v \in \{0, 1\}$ if `b` not already contains v
  - this operation is called *modify-and-test*
- the third case generalizes this to arbitrary values
  - this operation is called *compare-and-swap*

$\rightsquigarrow$ use as building blocks for algorithms that can *fail*

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.
Common usage pattern for *compare and swap*:

1. read the initial value in i into k (using memory barriers)
2. calculate a new value $j = f(k)$
3. update i to j if $i = k$ still holds
4. go to first step if $i \neq k$ meanwhile

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.
Common usage pattern for *compare and swap*:

1. read the initial value in i into k (using memory barriers)
2. calculate a new value $j = f(k)$
3. update i to j if $i = k$ still holds
4. go to first step if $i \neq k$ meanwhile

⚠ note: $i = k$ must imply that no thread has updated i

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.
Common usage pattern for *compare and swap*:

1. read the initial value in i into k (using memory barriers)
2. calculate a new value $j = f(k)$
3. update i to j if $i = k$ still holds
4. go to first step if $i \neq k$ meanwhile

⚠ note: $i = k$ must imply that no thread has updated i

$\leadsto$ general recipe for *lock-free* algorithms

- given a compare-and-swap operation for n bytes
- try to group variables for which an invariant must hold into n bytes
- read these bytes atomically
- calculate a new value
- perform a compare-and-swap operation on these n bytes

Lock-Free Algorithms

If a *wait-free* implementation is not possible, a *lock-free* implementation might still be viable.
Common usage pattern for *compare and swap*:

1. read the initial value in i into k (using memory barriers)
2. calculate a new value $j = f(k)$
3. update i to j if $i = k$ still holds
4. go to first step if $i \neq k$ meanwhile

⚠ note: $i = k$ must imply that no thread has updated i

$\leadsto$ general recipe for *lock-free* algorithms

- given a compare-and-swap operation for n bytes
- try to group variables for which an invariant must hold into n bytes
- read these bytes atomically
- calculate a new value
- perform a compare-and-swap operation on these n bytes

$\leadsto$ calculating new value must be *repeatable*